



Pontifícia Universidade Católica de Campinas  
Faculdade de Engenharia de Computação -  
FECOMP

Sistemas Operacionais B – Relatório Experimento 1  
Módulo de criptografia do kernel Linux

Beatriz Morelatto Lorente RA: 18071597

Cesar Marrote Manzano RA: 18051755

Fabricio Silva Cardoso RA: 18023481

Pedro Ignácio Trevisan RA: 18016568

## **Sumário**

<b>1. Introdução.....</b>	<b>3</b>
<b>2. Algoritmos de criptografia utilizados.....</b>	<b>4</b>
<b>3. Programa de teste.....</b>	<b>5</b>
<b>4. Módulo de criptografia.....</b>	<b>6</b>
<b>5. Compilação dos programas.....</b>	<b>11</b>
<b>6. Testes e resultados.....</b>	<b>12</b>
<b>7. Conclusão.....</b>	<b>17</b>

## **Introdução**

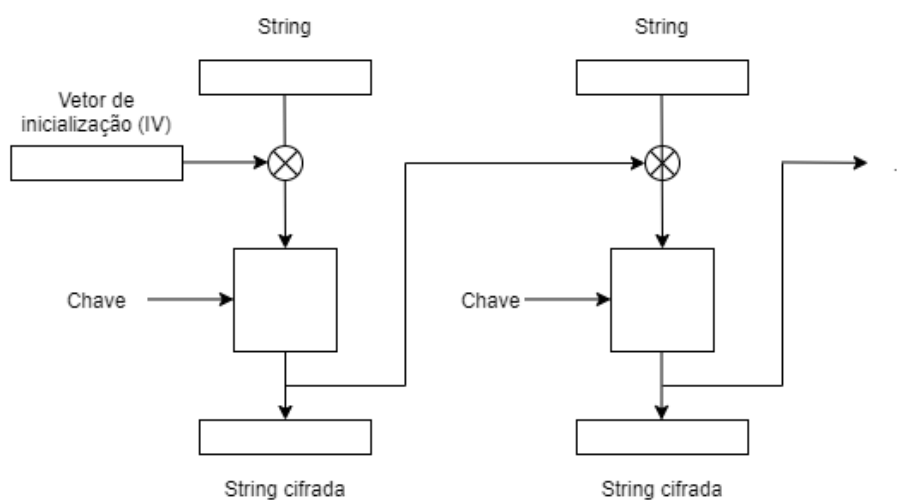
O experimento desenvolvido pretende demonstrar os passos feitos para o desenvolvimento de um módulo de criptografia do kernel Linux. O módulo é responsável por cifrar, decifrar ou calcular o resumo criptográfico (hash) de uma string fornecida pelo usuário. Para cifrar e decifrar uma string foi utilizado o algoritmo AES em modo CBC e para o resumo criptográfico o algoritmo SHA1. Para o teste do módulo foi criado um programa em espaço do usuário que se conecta com o dispositivo, enviando as requisições necessárias e exibindo o resultado da criptografia.

## Algoritmos de criptografia utilizados

Para cifrar e decifrar a string fornecida pelo usuário foi utilizado o algoritmo AES em modo CBC.

Primeiramente é necessário entender como o algoritmo AES (Advanced Encryption Standard) funciona. O algoritmo consiste em uma criptografia simétrica de blocos de tamanho fixo (utilizamos blocos de 16 bytes, 128 bits). O algoritmo usa uma chave para cifrar e decifrar os blocos e esta também tem 16 bytes.

No modo de criptografia CBC, para cada bloco de texto é aplicado a função XOR com o bloco anterior, garantindo que cada bloco seja dependente um do outro. Para o primeiro bloco, é aplicado um vetor de inicialização, garantindo que cada operação de criptografia seja única. O esquema do algoritmo é representado abaixo:

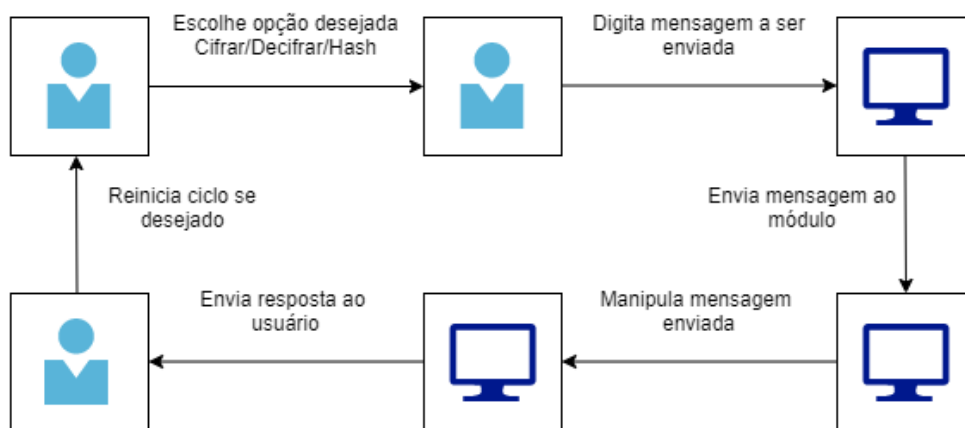


*Figura 1 - Esquema do modo de criptografia CBC*

Já para o resumo criptográfico foi utilizado o algoritmo SHA1. Consiste em uma função de dispersão criptográfica, considerado bem seguro. O algoritmo produz uma mensagem de dispersão com 20 bytes, que é conhecida como o resumo da mensagem original.

## Programa de teste

Para testarmos o experimento foi feito um programa em espaço de usuário que se comunica com o módulo. O esquema abaixo representa como essa comunicação é feita.



*Figura 2 - Esquema do funcionamento do programa de teste*

Primeiramente é escolhida a operação que se deseja fazer, ou seja, cifrar, decifrar ou calcular o resumo criptográfico de uma mensagem. Após a escolha o usuário digita qual a mensagem que será enviada ao módulo. Para saber qual operação será feita, foi concatenado as letras 'c' (cifrar), 'd' (decifrar) ou 'h' (hash), no final da mensagem, dependendo da escolha do usuário. Desse modo o módulo pode aplicar adequadamente o algoritmo em cima de uma mensagem, basta ler apenas a última posição da string para isso. Após manipular a mensagem (mais detalhes sobre essa manipulação serão discutidas mais para frente), o módulo devolve a mesma para o programa de teste e este se encarrega de imprimir as informações necessárias. Todo o ciclo pode ser feito novamente caso o usuário deseje.

## Módulo de criptografia

O funcionamento do módulo de criptografia é bastante simples, ele apenas manipula uma mensagem e envia para o programa de teste. Porém, a manipulação da mensagem não é um processo fácil e diversos erros podem ser cometidos na hora de sua implementação. O esquema a seguir mostra como o módulo funciona.

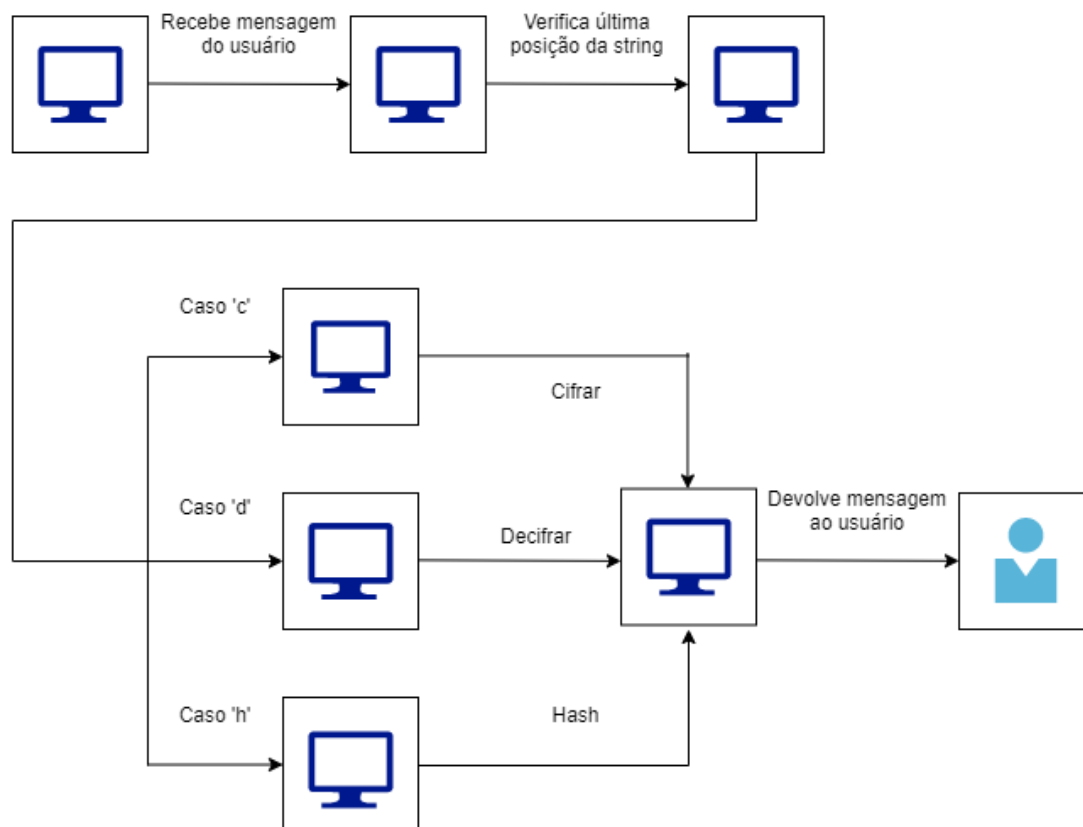


Figura 3 - Esquema do funcionamento do módulo de criptografia

Antes de prosseguir com a explicação sobre os algoritmos, é importante destacar alguns detalhes da comunicação entre os programas.

### O uso de mutex para a comunicação

Para que não houvesse nenhum problema de sincronização, foi necessário o uso de mutex. Com esse detalhe, foi possível bloquear um processo em espaço de usuário, evitando que mais de um processo fizesse uma requisição para o módulo de criptografia. O mutex é bloqueado quando o programa de teste tenta acessar (ler) o módulo e é liberado quando este fecha, ou seja, quando devolve uma mensagem. Para mais informações sobre o uso de mutex e a comunicação entre os programas, basta acessar o link a seguir: <http://derekmolloy.ie/writing-a-linux-kernel-module-part-2-a-character-device>. O esquema abaixo exemplifica o uso do mutex em nosso programa.

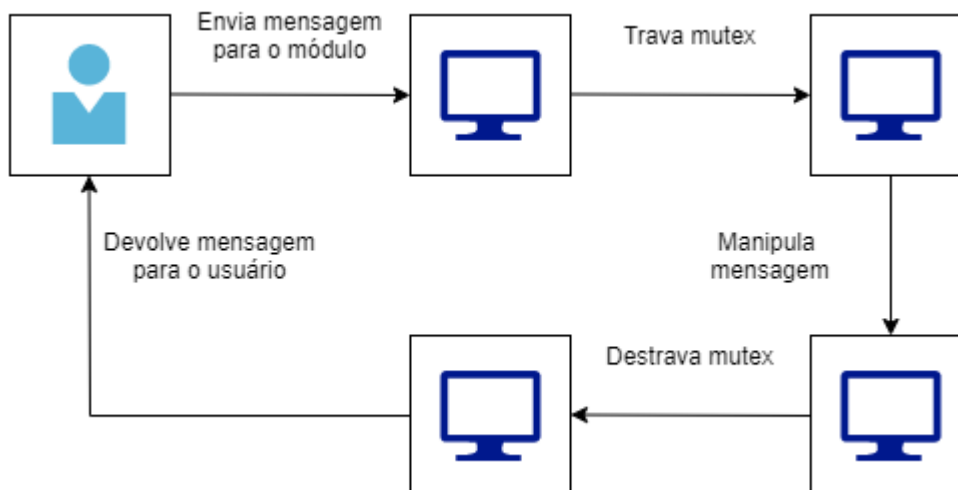


Figura 4 - Esquema do funcionamento do mutex

### Detalhes de implementação das operações de cifrar e decifrar

Como citado anteriormente, o programa é capaz de fazer operações de cifrar e decifrar uma mensagem. O link a seguir, da documentação da API do kernel linux, contém explicações e exemplos de código de criptografia: <https://www.kernel.org/doc/html/v4.12/crypto/index.html>. Alguns detalhes da implementação dos algoritmos serão discutidos abaixo.

Primeiramente, é importante lembrar que para essas operações, foi usado o algoritmo AES em modo CBC. Portanto é necessário se utilizar de uma chave e um vetor de inicialização. Os valores desses componentes foi passado como parâmetro ao módulo e fixado com '0123456789ABCDEF' em ambos os casos.

O primeiro passo para se implementar a função para cifrar(encrypt) e decifrar(decrypt), é dizer para o módulo qual algoritmo será usado e setar a estrutura de dados que será usada na operação. A imagem a seguir mostra como isso é feito.

```

/* Allocate a cipher handle for an skcipher */
skcipher = crypto_alloc_skcipher("cbc(aes)", 0, 0);
if (IS_ERR(skcipher))
{
    pr_info("could not allocate skcipher handle\n");
    return PTR_ERR(skcipher);
}

/* Allocate the request data structure that must be used with the skcipher encrypt and decrypt API calls */
req = skcipher_request_alloc(skcipher, GFP_KERNEL);
if (!req)
{
    pr_info("could not allocate skcipher request\n");
    ret = -ENOMEM;
    goto out;
}

skcipher_request_set_callback(req, CRYPTO_TFM_REQ_MAY_BACKLOG, test_skcipher_cb, &sk.result);
  
```

Figura 5 - Código da primeira parte de implementação para cifrar e decifrar

Com isso feito, é necessário requisitar um espaço em memória para alocar a chave e o vetor de inicialização. Como o tamanho de cada um é fixo em 16 bytes, basta alocar esse tamanho para ambos. A imagem a seguir mostra como isso é feito.

```
key_encrypt = vmalloc(16);

strcpy(key_encrypt, key);

if (crypto_skcipher_setkey(skcipher, key_encrypt, 16))
{
    pr_err("fail setting key");
    goto out;
}

/* ===== */

iv_encrypt = vmalloc(16);

if (!iv_encrypt)
{
    pr_err("could not allocate iv vector\n");
    ret = -ENOMEM;
    goto out;
}

strcpy(iv_encrypt, iv);
```

*Figura 6 - Código da segunda parte de implementação para cifrar e decifrar*

Também é necessário fazer o mesmo processo para a mensagem que é enviada, porém note que o espaço alocado para a mensagem é correspondente ao tamanho da mesma.

```
scratchpad = vmalloc(messageLength);
if (!scratchpad)
{
    pr_info("Could not allocate scratchpad\n");
    goto out;
}

memcpy(scratchpad, message, messageLength);
```

*Figura 7 - Código da primeira parte de implementação para cifrar e decifrar*

Por fim, basta setar os dados e chamar a função para cifrar ou decifrar a mensagem.



```

/* Setando struct */
sk.skcipher = skcipher;
sk.req = req;

/* Cifrar / Encrypt */
sg_init_one(&sk.sg, scratchpad, 16);
skcipher_request_set_crypt(req, &sk.sg, &sk.sg, 16, iv_encrypt);
init_completion(&sk.result.completion);

rc = crypto_skcipher_encrypt(req);

if (rc)
{
    pr_info("skcipher encrypt returned with %d result %d\n", rc, sk.result.err);
    goto out;
}

init_completion(&sk.result.completion);

result = sg_virt(&sk.sg);

strcpy(message, result);
printk("=====");
print_hex_dump(KERN_DEBUG, "Result Data Encrypt: ", DUMP_PREFIX_NONE, 16, 1, result, 16, true);
printk("=====");

```

*Figura 8 - Código da terceira parte de implementação para cifrar*

```

/* ===== */

/* Setando struct */
sk.skcipher = skcipher;
sk.req = req;

/* Decifrar / Decrypt */
sg_init_one(&sk.sg, scratchpad, 16);
skcipher_request_set_crypt(req, &sk.sg, &sk.sg, 16, iv_decrypt);
init_completion(&sk.result.completion);

rc = crypto_skcipher_decrypt(req);

if (rc)
{
    pr_info("skcipher encrypt returned with %d result %d\n", rc, sk.result.err);
    goto out;
}

init_completion(&sk.result.completion);

result = sg_virt(&sk.sg);
strcpy(message, result);

printk("=====");
print_hex_dump(KERN_DEBUG, "Result Data Decrypt: ", DUMP_PREFIX_NONE, 16, 1, result, 16, true);
printk("=====");

```

*Figura 9 - Código da terceira parte de implementação para decifrar*

## Detalhes de implementação da operação de hash

Assim como as funções de cifrar e decifrar, a função de hash também é simples de implementar. Novamente utilizamos o link da documentação da API de criptografia do kernel Linux para nos auxiliar na implementação desta função: <https://www.kernel.org/doc/html/v4.12/crypto/index.html>.

O primeiro passo para a implementação é requisitar o algoritmo que será utilizado (no caso SHA1). Depois é necessário alocar um espaço em memória para alocar as estruturas que serão usadas e logo depois setar as mesmas. Também é necessário alocar um espaço para armazenar o resultado. Após os passos, basta chamar a função de hash com os parâmetros necessários.

```
static int hash(char *message, int messageLength)
{
    struct shash_desc *shash;
    struct crypto_shash *req;
    char *result = NULL;
    int ret;

    req = crypto_alloc_shash("sha1", 0, 0);
    shash = vmalloc(sizeof(struct shash_desc));
    if (!shash)
        goto out;

    shash->tfm = req;
    shash->flags = 0x0;

    result = vmalloc(SHA1_SIZE);
    if (!result)
        goto out;

    ret = crypto_shash_digest(shash, message, messageLength, result);
    strcpy(message, result);

    /* ===== */

out:
    if (req)
        crypto_free_shash(req);
    if (shash)
        vfree(shash);
    if (result)
        vfree(result);

    return 0;
}
```

*Figura 10 - Código da implementação da função hash*

## Compilação dos resultados

Para que os programas fossem compilados foi feito um arquivo Makefile para que ambos fossem compilados juntos. A imagem abaixo mostra o arquivo.

```
obj-m+=cryptomodule.o

all:
    make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) modules
    sudo insmod cryptomodule.ko key="0123456789ABCDEF" iv="0123456789ABCDEF"
    $(CC) cryptoModuleTest.c -o test
    sudo ./test

clean:
    sudo rmmod cryptomodule
    sudo rm test
    make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) clean
```

*Figura 11 - Arquivo de compilação dos programas*

Quando é digitado o comando 'make all' no terminal do Linux, ambos os programas são compilados. Ao rodar o comando 'make clean', todos os arquivos objetos e dependências são removidos do sistema.

## Testes e Resultados

### Cifragem e Decifragem

Para os testes foram usadas as seguintes strings:

- bom
- pedro
- cesar
- hello world
- hello world 2

```
Digite a mensagem para ser cifrada: bom
Pressione ENTER para ler a resposta

String enviada: bom
Mensagem cifrada: 33C99E8FE108D55E12181DE8A19BA352
Pressione ENTER para continuar
```

*Figura 12 - Primeira string cifrada*

```
Digite a mensagem para ser decifrada: 33C99E8FE108D55E12181DE8A19BA352
Pressione ENTER para ler a resposta

Mensagem Enviada: 33C99E8FE108D55E12181DE8A19BA352
Mensagem Decifrada: 626F6D8FE108D55E12181DE8A19BA352
bom
Pressione ENTER para continuar
```

*Figura 13 - Primeira string decifrada*

```
Digite a mensagem para ser cifrada: pedro
Pressione ENTER para ler a resposta

String enviada: pedro
Mensagem cifrada: C7131D5524E9688D6BC1A040B16A126A
Pressione ENTER para continuar
```

*Figura 14 - Segunda string cifrada*

```
Digite a mensagem para ser decifrada: C7131D5524E9688D6BC1A040B16A126A
Pressione ENTER para ler a resposta
Mensagem Enviada: C7131D5524E9688D6BC1A040B16A126A
Mensagem Decifrada: 706564726FE9688D6BC1A040B16A126A
pedro
Pressione ENTER para continuar
```

*Figura 15 - Segunda string decifrada*

```
Digite a mensagem para ser cifrada: cesar
Pressione ENTER para ler a resposta
String enviada: cesar
Mensagem cifrada: 01C950FA7ABE1BD40BA6EDD2765A4187
Pressione ENTER para continuar
```

*Figura 16 - Terceira string cifrada*

```
Digite a mensagem para ser decifrada: 01C950FA7ABE1BD40BA6EDD2765A4187
Pressione ENTER para ler a resposta
Mensagem Enviada: 01C950FA7ABE1BD40BA6EDD2765A4187
Mensagem Decifrada: 6365736172BE1BD40BA6EDD2765A4187
cesar
Pressione ENTER para continuar
```

*Figura 17 - Terceira string decifrada*

```
Digite a mensagem para ser cifrada: hello world
Pressione ENTER para ler a resposta
String enviada: hello world
Mensagem cifrada: 76E2365234FCCDA2C5F1980F94DF94C9
Pressione ENTER para continuar
```

*Figura 18 - Quarta string cifrada*

```
Digite a mensagem para ser decifrada: 76E2365234FCCDA2C5F1980F94DF94C9
Pressione ENTER para ler a resposta

Mensagem Enviada: 76E2365234FCCDA2C5F1980F94DF94C9
Mensagem Decifrada: 68656C6C6F20776F726C640F94DF94C9
hello world

Pressione ENTER para continuar
```

*Figura 19 - Quarta string decifrada*

```
Digite a mensagem para ser cifrada: hello world 2
Pressione ENTER para ler a resposta

String enviada: hello world 2
Mensagem cifrada: 8116EC66177463F08830C71183A039C9

Pressione ENTER para continuar
```

*Figura 20 - Quinta string cifrada*

```
Digite a mensagem para ser decifrada: 8116EC66177463F08830C71183A039C9
Pressione ENTER para ler a resposta

Mensagem Enviada: 8116EC66177463F08830C71183A039C9
Mensagem Decifrada: 68656C6C6F20776F726C642032A039C9
hello world 2

Pressione ENTER para continuar
```

*Figura 21 - Quinta string decifrada*

## Hash

Para os testes foram usadas as seguintes strings:

- hello world
- testando o programa
- ordem e progresso
- um ninho de mafagafos tinha sete mafagafinhos

Para confirmar o resultado foi utilizado o site <http://www.sha1-online.com/>.

## SHA1 and other hash functions online generator

hello world hash

sha-1

Result for sha1: 2aae6c35c94fcfb415dbe95f408b9ce91ee846ed

```
root@cesar-VirtualBox: /home/cesar/Documentos/CryptoDeviceDriver
Digite a mensagem desejada para calcular o resumo criptográfico: hello world
Mensagem enviada: hello world
Pressione ENTER para ler a resposta
Resumo criptográfico: 2AAE6C35C94FCFB415DBE95F408B9CE91EE846ED
Pressione ENTER para continuar
```

Figura 22 - Hash primeira string

## SHA1 and other hash functions online generator

testando o programa hash

sha-1

Result for sha1: e862a50c6e1d1a81cf8c48bbbf07d61f47e4fc76

```
root@cesar-VirtualBox: /home/cesar/Documentos/CryptoDeviceDriver
Digite a mensagem desejada para calcular o resumo criptográfico: testando o programa
Mensagem enviada: testando o programa
Pressione ENTER para ler a resposta
Resumo criptográfico: E862A50C6E1D1A81CF8C48BBBF07D61F47E4FC76
Pressione ENTER para continuar
```

Figura 23 - Hash segunda string

## SHA1 and other hash functions online generator

ordem e progresso hash

sha-1

Result for sha1: 7e4fcf7de381c65f28ab5446ac3dd4fc152d3420

```
root@cesar-VirtualBox: /home/cesar/Documentos/CryptoDeviceDriver
Digite a mensagem desejada para calcular o resumo criptográfico: ordem e progresso
Mensagem enviada: ordem e progresso
Pressione ENTER para ler a resposta
Resumo criptográfico: 7E4FCF7DE381C65F28AB5446AC3DD4FC152D3420
Pressione ENTER para continuar
```

Figura 24 - Hash terceira string

## SHA1 and other hash functions online generator

um ninho de mafagafos tinha sete mafagafinhos hash

sha-1

Result for sha1: 53c825e7c6961948f4d74fe115d573473bba2d1f

```
root@cesar-VirtualBox: /home/cesar/Documentos/CryptoDeviceDriver
Digite a mensagem desejada para calcular o resumo criptográfico: um ninho de maf
agafos tinha sete mafagafinhos

Mensagem enviada: um ninho de mafagafos tinha sete mafagafinhos

Pressione ENTER para ler a resposta

Resumo criptográfico: 53C825E7C6961948F4D74FE115D573473BBA2D1F

Pressione ENTER para continuar
```

*Figura 25 - Hash quarta string*



## **Conclusão**

Com o experimento foi possível compreender melhor como um módulo de linux funciona. Além disso foi possível entender como implementar um programa em espaço de usuário se comunica com um módulo de kernel.

Além disso também foi possível obter novos conhecimentos na área de criptografia, conhecendo os algoritmos AES em modo CBC e SHA1. São algoritmos que são disponibilizados pela API de criptografia do kernel e de fácil implementação.